UNITED STATES PATENT APPLICATION OF:
NIKOLAOS KOUDAS, DIVESH SRIVASTAVA, PANAGIOTIS IPEIROTIS AND LUIS GRAVANO FOR:
TEXT JOINS FOR DATA CLEANSING AND INTEGRATION IN A RELATIONAL DATABASE MANAGEMENT SYSTEM

CROSS-REFERENCE TO RELATED APPLICATIONS

This application claims priority of U.S. Provisional Application No. 60/464,101, filed on, April 21, 2003, which is incorporated by reference herein.

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates to a method for identifying potential string matches across relations within a relational database management system.

2. Description of Related Art

Integrating information from a variety of homogeneous or heterogeneous data sources is a problem of central interest. With the prevalence of the web, a number of emerging applications, such as catalog integration and warehousing of web data (e.g., job advertisements and announcements), face data integration at the very core of their operation. Corporations increasingly request to obtain unified views of their information (e.g., customers, employees, products, orders, suppliers), which makes data integration of critical importance.

Data integration also arises as a result of consolidation (e.g., mergers and takeovers) both at inter- as well as intra-corporation levels. Consider a large service provider corporation offering a variety of services. The corporation records a multitude of information per customer (such as name and address) in corporate databases. This information often excludes unique global identifiers (such as Social Security Number) in accordance with corporate or federal policies. Customers subscribe to one or more services. Due to a variety of reasons -including the specifics of the business model and organization boundaries different information systems with customer

information may be maintained for each service. Let *Rl* and *R2* be two relations recording the name and address of customers of two services. In the presence of global identifiers, a straightforward join between R_1 and R_2 on the unique identifier would match customers across both services. In the absence of global identifiers, deducing whether two or more customers represent the same entity turns out to be a challenging problem, since one has to cope with mismatches arising from:

erroneous information (for example, typing mistakes when customer information is acquired),

missing or incomplete information,

differences in information "formatting" due to the lack of standard conventions (e.g., for addresses)

or a combinations of any of the preceding errors.

For example, observing the name attribute instances "AT&T Research" of relation R_1, and "ATT Research Labs" (or "AT&T Labs Research") of R_2, can we deduce that they correspond to the same entity. Are "AT&T Research" and "AT&T Research Labs" more likely to correspond to the same entity than "AT&T Research" and "AT&T Labs Research"? If we consider the additional address field, are the instances ("AT&T Research", "Florham Park"), ("AT&T Research Labs", "Florham Park NJ") more likely to correspond to the same entity than ("AT&T Research"," Florham Park"), ("AT&T Labs Research"," Menlo Park CA")?Any attempt to address the integration problem has to specify a measure that effectively quantifies "closeness" or "similarity" between string attributes. Once this measure is specified, there is a clear need for algorithms that efficiently process the data sources and join them to identify all pairs of strings (or sets of strings) that are sufficiently similar to each other. Furthermore, it is desirable to

perform such a join, which we refer to as a text-join, within an unmodified relational database management system (RDBMS),which is where the data is likely to reside The present invention defines text-joins using the cosine similarity metric to quantify string similarity, as well as defines algorithms to process text joins efficiently in an RDBMS..

SUMMARY OF THE INVENTION

The present invention provides a system for string matching across multiple relations in a relational database management system comprising generating a set of strings from a set of characters, decomposing each string into a subset of tokens, establishing at least two relations within the strings, establishing a similarity threshold for the relations, sampling the at least two relations, correlating the relations for the similarity threshold and returning all of the tokens which meet the criteria of the similarity threshold.

BRIEF DESCRIPTION OF THE DRAWINGS

The various features, objects, benefits, and advantages of the present invention will become more apparent upon reading the following detailed description of the preferred embodiment(s) along with the appended claims in conjunction with the drawings, wherein like reference numerals identify like components throughout, and:

FIG. 1 depicts an example of an SQL statement according to the present invention.

FIG. 2 depicts an example of the algorithm according to the present invention for computing the exact value of a particular relation.

FIG. 3 depicts an example of the algorithm according to the present invention for computing a sample relation.

FIG. 4 depicts an alternate example of the algorithm according to the present invention for computing a sample relation.

FIG. 5 depicts an example of the SQL algorithm according to the present invention for computing the weight and thresholding steps.

FIG. 6 depicts an example of the algorithm according to the present invention for a symmetric sampling-based text join.

FIG. 7 depicts an alternate example of the algorithm according to the present invention for a symmetric sampling-based text join.

FIG. 8a and 8b are graphs of two data sets for relations according to the present invention.

FIG. 9a, 9b and 9c depict graphs of the average precision and recall of different algorithms according to the present invention.

FIG. 10a, 10b and 10c depict graphs of the average precision and recall of different algorithms according to the present invention.

FIG. 11a and 11b depict graphs of the average precision and recall of different algorithms according to the present invention.

FIG. 12a, 12b, 12 c and 12d depict graphs of the average execution times of different algorithms according to the present invention.

DETAILED DESCRIPTION OF THE INVENTION

In describing this invention there is first provided a notation and background for text joins, which we follow with a formal definition of the problem on which we focus in this paper. We denote with Σ^* the set of all strings over an alphabet . Σ Each string in Σ^* can be decomposed into a collection of atomic "entities" that we generally refer to as tokens. What constitutes a token can be defined in a variety of ways. For example, the tokens of a string could simply be defined as the "words" delimited by special characters that are treated as "separators" (e.g., " ") alternatively, the tokens of a string could correspond to all of its *q-grams*, which are overlapping substrings of exactly q consecutive characters, for a given q. In the following discussion, the term token is treated as generic, as the particular choice of token is orthogonal to the design of our algorithms.

Let R_1 and R_2 be two relations with the same or different schemas and attributes. To simplify our discussion and notation we assume, without loss of generality, that we assess similarity between the entire sets of attributes of R_1 and R_2. Our discussion extends to the case of arbitrary subsets of attributes in a straightforward way. Given tuples:

$$t_1 \in R_1 \text{ and } t_2 \in R_2,$$

we assume that the values of their attributes are drawn from Σ^*. We adopt the vector-space retrieval model to define the textual similarity between t_1 and t_2.

Let D be the (arbitrarily ordered) set of all unique tokens present in all values of attributes of both R_1 and R_2. According to the vector-space retrieval model, we conceptually map each tuple

$$t \in R_i$$

to a vector

$$v_t \in \Re^{|D|}$$

The value of the *j-th* component $v_t(j)$ of v_t is a real number that corresponds to the weight of the *j-th* token of D in v_t. Drawing an analogy with information retrieval terminology, D is the set of all terms and v_t is a document weight vector.

Rather than developing new ways to define the weight vector v_t for a tuple

$$t \in R_i,$$

we exploit an instance of the well-established *tf.idf* weighting scheme from the information retrieval field. (*tf.idf* stands for "term frequency, inverse document frequency.") Our choice is further supported by the fact that a variant of this general weighting scheme has been successfully used for our task by Cohen's WHIRL system. Given a collection of documents C, a simple version of the *tf.idf* eight for a term w and a document d is defined as;

$tf_w \log(idf_w)$,

where

tf_w is the number of times that *w* appears in document *d* and

idf_w,

is

$\frac{|C|}{n_w}$, where n_w is the number of documents in the collection *C* that contain term *w*. The *tf.idf* weight for a term w in a document is high if *w* appears a large number of times in the document and *w* is a sufficiently "rare" term in the collection (i.e., if *w*'s discriminatory power in the collection is potentially high). For example, for a collection of company names, relatively infrequent terms such as "AT&T" or "IBM" will have higher *idf* weights than more frequent terms such as "Inc."

For our problem, the relation tuples are our "documents," and the tokens in the textual attribute of the tuples are our "terms." Consider the j-th token w in *D* and a tuple *t* from relation R_i. Then *tfw* is the number of times that w appears in *t*. Also, *idfw* is:

$\frac{|R_i|}{n_w}$,

where n_w is the total number of tuples in relation R_i that contain token *w*. The *tf.idf* weight for token *w* in tuple;

$t \in R_i$ is $v_t(j) = tf_w \log(idf_w)$

To simplify the computation of vector similarities, we normalize vector v_t to unit length in the Euclidean space after we define it (the resulting weights corresponds to the impact of the terms).

Note that the weight vectors will tend to be extremely sparse for certain choices of tokens; we shall seek to utilize this sparseness in our proposed techniques

Definition 1 (Cosine Similarity) *Given tuples $t_1 \in R_1$ and $t_2 \in R_2$, let v_{t_1} and v_{t_2} be their corresponding normalized weight vectors and D is the set of all tokens in R_1 and R_2. The* cosine similarity *(or just* similarity, *for brevity) of v_{t_1} and v_{t_2} is defined as:*

$$sim(v_{t_1}, v_{t_2}) = \sum_{j=1}^{|D|} v_{t_1}(j) v_{t_2}(j)$$

Since vectors are normalize his measure corresponds to the cosine of the angle between vectors v_{i1} and v_{i2}, and has values between 0 and 1. The intuition behind this scheme is that the magnitude of a component of a vector expresses the relative "importance" of the corresponding token in the tuple represented by the vector. Intuitively, two vectors are similar if they share many important tokens. For example, the string "ACME" will be highly similar to "ACME Inc," since the two strings differ only on the token "Inc," which appears in many different tuples, and hence has low weight. On the other hand, the strings "IBM Research" and "AT&T Research" will have lower similarity as they share only one relatively common term. The following join between relations R_1 and R_2 brings together the tuples from these relations that are "sufficiently close" to each other according to a user-specified similarity threshold;

ϕ:

Definition 2 (Text-Join) *Given two relations R_1 and R_2, together with a similarity threshold $0 \leq \phi \leq 1$, the* text-join *$R_1 \bowtie_\phi R_2$ returns all pairs of tuples (t_1, t_2) such that:*

- *$t_1 \in R_1$ and $t_2 \in R_2$, and*
- *$sim(v_{t_1}, v_{t_2}) \geq \phi$.*

This text-join "correlates" two relations for a given similarity threshold

ϕ:

It can be easily modified to correlate arbitrary subsets of attributes of the relations. In this paper, we address the problem of computing the text-join of two relations efficiently and within an

unmodified RDBMS: Problem 1 Given two relations R_1 and R_2, together with a similarity threshold $0 \leq \phi \leq 1$, we want to efficiently compute (an approximation of) the text-join

$R_1 \bowtie_\phi R_2$

using "vanilla" SQL in an unmodified RDBMS. We first describe our methodology for deriving, in a preprocessing step, the vectors corresponding to each tuple of relations R_1 and R_2 using relational operations and representations. We then present our sampling-based solution for efficiently computing the text join of the two relations using standard SQL in an RDBMS

Creating Weight Vectors for Tuples In .this section, we describe how we define auxiliary relations to represent tuple weight vectors. In the following section, we develop a sampling-based technique to compute the text-join of two relations starting with the auxiliary relations that we define next. As in the previous section, it is assumed that we want to compute the text-join

$R_1 \bowtie_\phi R_2$

of two relations R_1 and R_2. D is the ordered set of all the tokens that appear in R_1 and R_2. We use SQL expressions to create the weight vector associated with each tuple in the two relations. Since for some choice of tokens each tuple is expected to contain only a few of the tokens in D, the associated weight vector is sparse. We exploit this sparseness and represent the weight vectors by storing only the tokens with non-zero weight. Specifically, for a choice of tokens (e.g., words or q-grams), we create the following relations for a relation R_i:

- *RiTokens(tid, token)*: Each tuple (tid, w) is associated with an occurrence of token w in the R_i tuple with id *tid*. This relation is populated by inserting exactly one tuple (tid, w) for each occurrence of token w in a tuple of R_i with tuple id *tid*. This relation can be implemented in pure SQL and the implementation varies with the choice of tokens. (See [?] for an example on how to create this relation when q-grams are used as tokens.)

- *RiIDF(token, idf)*: A tuple (w, idf_w) indicates that token w has inverse document frequency idf_w (Section 2) in relation R_i. The SQL statement to populate relation *RiIDF* is shown in Figure 1(a). This statement relies on a "dummy" relation *RiSize(size)* (Figure 1(f)) that has just one tuple indicating the number of tuples in R_i.

- *RiTF(tid, token, tf)*: A tuple (tid, w, tf_w) indicates that token w has term frequency tf_w (Section 2) for R_i tuple with tuple id *tid*. The SQL statement to populate relation *RiTF* is shown in Figure 1(b).

- *RiLength(tid, len)*: A tuple (tid, l) indicates that the weight vector associated with R_i tuple with tuple id *tid* has a Euclidean norm of l. (This relation is used for normalizing weight vectors.) The SQL statement to populate relation *RiLength* is shown in Figure 1(c).

- *RiWeights(tid, token, weight)*: A tuple (tid, w, n) indicates that token w has normalized weight n in R_i tuple with tuple id *tid*. The SQL statement to populate relation *RiWeights* is shown in Figure 1(d). This relation materializes a compact representation of the final weight vector for the tuples in R_i.

- *RiSum(token, total)*: A tuple (w, t) indicates that token w has a total added weight t in relation R_i, as indicated in relation *RiWeights*. These numbers are used during sampling (see Section 4). The SQL statement to populate relation *RiSum* is shown in Figure 1(e).

Given two relations R_1 and R_2, we can use the SQL statements in Figure 1 to generate relations *R1Weights* and *R2Weights* with a compact representation of the weight vector for the R_1 and R_2 tuples. Only the non-zero *tf.idf* weights are stored in these tables. The space overhead introduced by these tables is moderate. Since the size of R*iSum* is bounded by the size of *RiWeights*, we just analyze the space requirements for *RiWeights*. Consider the case where q-grams are the tokens of choice. (As we will see, a good value is $q = 3$.) Then each tuple $R_i.t_j$ of relation *Ri* can contribute up to approximately;

$|R_i.t_j|$

q-grams to relation *RiWeights*, where

$|R_i.t_j|$

is the number of characters in $R_i.t_j$. Furthermore, each tuple in *RiWeights* consists of a tuple id *tid,* the actual token (i.e,, q-gram in this case), and its associated weight. Then, if C bytes are needed to represent *tid* and weight, the total size of relation *RiWeights* will not exceed;

$$\sum_{j=1}^{|R_i|}(C + q) \cdot |R_i.t_j| = (C+q) \cdot \sum_{j=1}^{|R_i|} |R_i.t_j|,$$

which is a (small) constant times the size of the original table *Ri.* If words are used as the token of choice, then we have at most $\frac{|R_i.t_j|}{2}$ tokens per tuple in *Ri.* Also, to store the token attribute of *RiWeights* we need no more than one byte for each character in the $R_i.t_j$ tuples. Therefore, we can bound the size of *RiWeights* by $1 + \frac{C}{2}$ times the size of *Ri.* Again, in this case the space overhead is linear in the size of the original relation *R.* Given the relations *R1Weights* and *R2Weights*, a baseline approach to compute:

$R_1 \bowtie_\phi R_2$

is shown in Figure 2.

This SQL statement performs the text-join by computing the similarity of each pair of tuples and filtering out any pair with similarity less than the similarity threshold ϕ. This approach produces an exact answer to;

$R_1 \bowtie_\phi R_2$ for $\phi > 0$.

As will be described later, finding an exact answer with this approach is expensive, which motivates the sampling-based technique that we describe next.

The result of $R_1 \bowtie_\phi R_2$ only contains pairs of tuples from R_1 and R_2 with similarity ϕ or higher. Usually we are interested in high values for threshold ϕ, which should result in only a few tuples from R_2 typically matching each tuple from R_1. The baseline approach in Figure 2,

however, calculates the similarity of all pairs of tuples from R_1 and R_2 that share at least one token. As a result, this baseline approach is inefficient: most of the candidate tuple pairs that it considers do not make it to the final result of the text-join. In this section, we present a sampling-based technique to execute text-joins efficiently, drastically reducing the number of candidate tuple pairs that are considered during query processing. Our sampling-based technique relies on the following intuition:

$$R_1 \tilde{\bowtie}_{\phi} R_2$$

could be computed efficiently if, for each tuple t_q of R_1, we managed to extract a sample from R_2 containing mostly tuples suspected to be highly similar to t_q. By ignoring the remaining (useless) tuples in R_2, we could approximate

$$R_1 \tilde{\bowtie}_{\phi} R_2$$

efficiently. The key challenge then is how to define a sampling strategy that leads to efficient text-join executions while producing an accurate approximation of the exact query results. The discussion of our technique is organized as follows:

• Similarity Sampling shows how to sample from R2, (unrealistically, but deliberately) assuming knowledge of all tuple-pair similarity values.

• Token Weighted Sampling shows how to estimate the tuple-pair similarity values by sampling directly from the tuple vectors of R_2.

• Finally, Practical Realization of Sampling describes an efficient algorithm for computing an approximation of the text-join.

Similarity Sampling

The description of our approach will rely on the following conceptual vector, which will never be fully materialized and which contains the similarity of a tuple tq from relation R_1 with each tuple of relation R_2:

$$V(t_q) = [sim(v_{t_q}, v_{t_1}), \ldots, sim(v_{t_q}, v_{t_i}), \ldots, sim(v_{t_q}, v_{t_{|R_2|}})]$$

When t_q is clear from the context, to simplify the notation we use; σ_i, as shorthand for $sim(v_{t_q}, v_{t_i})$.

Hence we have:

$$V(t_q) = [\sigma_1, \ldots, \sigma_i, \ldots, \sigma_{|R_2|}]$$

Intuitively, our techniques will efficiently compute an approximation of vector $V(t_q)$ for each tuple;

$t_q \in R_1$.

The approximation can then be used to produce a close estimate of;

$R_1 \bowtie_\phi R_2$.

Assume that $V(t_q)$ is already computed and available at hand (we will relax this requirement in the next section). We define;

$T_V(t_q)$ as the sum of all entries in;

$V(t_q)$ (i.e., $T_V(t_q)$ is the sum of the similarity of tuple t_q with each tuple $t_i \in R_2$):

$$T_V(t_q) = \sum_{i=1}^{|R_2|} \sigma_i$$

Now, consider taking a sample of some size S from the set of R_2 tuples;

$$\{t_1, \ldots, t_{|R_2|}\},$$

where the probability of picking;

$$t_i \text{ is } p_i = \frac{\sigma_i}{T_V(t_q)}$$

(i.e., the probability of picking t_i is proportional to the similarity of R_2 tuple t_i and our "fixed" R_1 tuple t_q). To get the S samples, we consider each tuple t_i S times. Let C_i be the number of times that t_i appears in the sample under this sampling strategy. We will show that;

$$\frac{C_i}{S} T_V(t_q)$$

provides an estimate of σ_i and we will establish a relationship between the sampling size S and the quality of estimation of σ_i. Specifically, the probability that t_i is included τ times in a sample of size S is;

$$P[C_i = \tau] = \binom{S}{\tau} p_i^{\tau} (1 - p_i)^{(S-\tau)}$$

In other words, each C_i is a Bernoulli trial with parameter pi and mean $S \cdot p_i$ Moreover, the C_i's are independent. According to the Hoeffding bounds, for n trials of binomial variable X with mean μ and for $0 < e < 1$, we know:

$$\begin{aligned} P[X - \mu > \epsilon n] &\leq e^{-2n\epsilon^2} \text{ and} \\ P[X - \mu < -\epsilon n] &\leq e^{-2n\epsilon^2} \end{aligned}$$

Substituting in the equations above;

$$X = C_i,\ n = S, \text{ and } \mu = S \cdot p_i, \text{ where } p_i = \frac{\sigma_i}{T_V(t_q)}:$$

$$P[\frac{C_i}{S}T_V(t_q) - \sigma_i > \epsilon T_V(t_q)] \leq e^{-2S\epsilon^2} \quad (1)$$

and

$$P[\frac{C_i}{S}T_V(t_q) - \sigma_i < -\epsilon T_V(t_q)] \leq e^{-2S\epsilon^2} \quad (2)$$

Thus, we can get arbitrarily close to each σ_i by choosing an appropriate sample size S.

Specifically, if we require the similarity estimation error;

$\epsilon T_V(t_q)$ to be smaller than δ_s, and the probability of error;

$e^{-2S\epsilon^2}$ be smaller than δ_p, we can solve the two inequalities;

$\epsilon T_V(t_q) \leq \delta_s$ and,

$e^{-2S\epsilon^2} \leq \delta_p$ to get a suitable sample size S:

$$S \geq \frac{\ln(\delta_p^{-1})}{2\delta_s^2} T_V(t_q)^2$$

The Sampling scheme that we described so far in this section is of course not useful in practice: If we knew $V(t_q)$, then we could just report all R_2 tuples with similarity;

$\sigma_i \geq \phi$.

In this section, it is described how to estimate the entries of $V(t_q)$, by sampling directly from the set of tokens of R_2. As discussed, the sampling strategy outlined above cannot be immediately realized for our problem, since $V(t_q)$ is not known a-priori. We now show how to perform sampling according to the values of $V(t_q)$ without computing $V(t_q)$ explicitly. Consider tuple $t_q \in R_1$ with its associated token weight vector;

v_{t_i}. We extract a sample of R_2 tuples of size S for tq -with no knowledge of $V(t_q)$ as follows:

• Identify each token j in t_q that has non-zero weight

$v_{t_q}(j), 1 \leq j \leq |D|$.

For each such token j, perform S Bernoulli trials over each;

$t_i \in \{t_1, \ldots, t_{|R_2|}\}$,

where the probability of picking t_i in a trial depends on the weight of token j in tuple

$t_q \in R_1$ and in tuple $t_i \in R_2$.

Specifically, this probability is;

$p_{ij} = \frac{v_{t_q}(j) \cdot v_{t_i}(j)}{Tv(t_q)}$. (We describe below how we can compute;

$Tv(t_q)$ efficiently without information about the individual entries σ_i of $V(t_q)$.)

Let C_i be the number of times that t_i appears in the sample of size S. It follows that:

Theorem 4.1 *The expected value of* $\frac{C_i}{S} \cdot Tv(t_q)$ *is* σ_i.

The proof of this theorem follows from an argument similar to that of Section 4.1 and from the observation that the mean of the process that generates C_i is $\frac{\sum_{j=1}^{|D|} v_{t_q}(j) v_{t_i}(j)}{Tv(t_q)} = \frac{\sigma_i}{Tv(t_q)}$.

Theorem 4.1 establishes that, given a tuple $t_q \in R_1$, we can obtain a sample of size S of tuples t_i such that the frequency C_i of tuple t_i can be used to approximate σ_i. We can then report

$\langle t_q, t_i \rangle$

as part of the answer $R_1 \bowtie_\phi R_2$ for each tuple $t_i \in R_2$ such that its estimated similarity with t_q (i.e., its estimated σ_i)is ϕ' or larger, where $\phi' = (1 - \epsilon)\phi$ is a slightly lower threshold, where ε is treated as a positive constant of less than 1, derived from Equations 1 and 2. An apparent problem of the sampling scheme proposed so far is the lack of knowledge of the value $Tv(t_q)^2$.

We show that this value can be easily calculated without knowledge of the individual values σ_i.of $V(t_q)$. First, we define *Sum(j)* as the total weight of the *j*-th token in relation;

R_2, $Sum(j) = \sum_{i=1}^{|R_2|} v_{t_i}(j)$. (These weights are kept in relation *R2Sum*.) Then, it is the case that:

$$T_V(t_q) = \sum_{i=1}^{|R_2|}\sum_{j=1}^{|D|} v_{t_q}(j)v_{t_i}(j) = \sum_{j=1}^{|D|} v_{t_q}(j)\sum_{i=1}^{|R_2|} v_{t_i}(j) = \sum_{j=1}^{|D|} v_{t_q}(j)Sum(j) \qquad (3)$$

Consequently, $Tv(t_q)$ can be easily computed from the values stored in *R2Sum* and in *R1Weights* that are already computed using the SQL statements of the previous section.

Given R_1, R_2 and a threshold ϕ, our discussion suggests the following strategy for the evaluation of the $R_1 \bowtie_\phi R_2$ text-join, in which we process one tuple $t_q \in R_1$ .at a time:

- Obtain an individual sample of size S from R_2 for t_q, using vector v_{t_q} to sample tuples of R_2 for each token with non-zero weight in v_{t_q}.
- If C_i is the number of times that tuple t_i appears in the sample for t_q, then use $\frac{C_i}{S}T_V(t_q)$ as an estimate of σ_i.
- Include tuple pair (t_q, t_i) in the text-join result only if $\frac{C_i}{S}T_V(t_q) > \phi'$ (or equivalently $C_i > \frac{S}{T_V(t_q)}\phi'$), and filter out the remaining R_2 tuples. We refer to this filter as *count filter*.

This strategy guarantees that identify all pairs of tuples with similarity above ϕ, with a desired probability, as long as we choose an appropriate sample size *S*. So far, the discussion has focused on obtaining an R_2 sample of size *S* individually for each tuple;

$t_q \in R_1$.

A naive implementation of this sampling strategy would then require a scan of relation R_2 for each tuple in R_1, which is clearly unacceptable in terms of performance. In the next section we describe how we perform the sampling with only one sequential scan of relation R_2.

Practical Realization of Sampling

As discussed so far, our sampling strategy requires extracting a separate sample from R_2 for each tuple in R_1. This extraction of a potentially large set of independent samples from R_2 (i.e., one per R_1 tuple)is of course inefficient, since it would require a large number of scans of the R_2table. In this section, we describe how we adapt the original sampling strategy so that it requires one single sample of R_2 and show how we use this sample to create an approximate answer for the text-join;

$R_1 \tilde{\bowtie}_{\phi} R_2$.

As we have seen in the previous section, for each tuple;

$t_q \in R_1$

we should sample a tuple t_i from R_2 in a way that depends on the $v_{t_q}(j) \cdot v_{t_i}(j)$ values. Since these values are different for each tuple of R_1, as straight forward implementation of this sampling strategy requires multiple samples of relation R_2. Here we describe an alternative sampling strategy that requires just one sample of R_2: First, we sample R_2using only the

$v_{t_i}(j)$

weights from the tuples t_i of R_2:, to generate a single sample of R_2:. Then, we use the single sample differently for each tuple t_q of R_1. Intuitively, we "weight" the tuples in the sample according to the weights

$v_{t_q}(j)$ of the t_q tuples of R_1. In particular, for a desired sample size S and a target similarity ϕ, we realize our sampling-based text-join;

$R_1 \tilde{\bowtie}_{\phi} R_2$

in three steps:.

1. **Sampling:** We sample the tuple ids i and the corresponding tokens from the vectors v_{t_i} for each tuple $t_i \in R_2$. We sample each token j from a vector v_{t_i} with probability $\frac{v_{t_i}(j)}{Sum(j)}$. We perform S trials, yielding approximately S samples for each token j.
2. **Weight:** For each $t_q \in R_1$ and for each token j with non-zero weight in v_{t_q}, scan the sample of R_2 and pick each tuple t_i with probability $\frac{v_{t_q}(j) \cdot Sum(j)}{T_V(t_q)}$. For each successful trial, add the corresponding tuple pair $\langle t_q, t_i \rangle$ to the candidate set.
3. **Thresholding:** After creating the candidate set, count the number of occurrences of each tuple pair $\langle t_q, t_i \rangle$. Add tuple pair $\langle t_q, t_i \rangle$ to the final result only if its frequency satisfies the count filter (Section 4.2).

Such a sampling scheme identifies tuples with similarity above ϕ from R_2 for each tuple in R_1. Observe for each;

$$t_q \in R_1$$

we obtain S samples in total choosing samples according to;

$\frac{v_{t_q}(j) v_{t_i}(j)}{T_V(t_q)}$ in expectation.

By sampling R_2 only once, the sample will be correlated. As we verify experimentally in the Experimental Evaluation of the present invention, this sample correlation has negligible effect on the quality of the join approximation. The proposed solution, as presented, is asymmetric in the sense that it uses tuples from one relation(R_1) to weight samples obtained from the other (R_2). The text-join problem, as defined, is symmetric and does not distinguish or impose an ordering on the operands (relations). Hence, the execution of the text-join $R_1 \bowtie_\phi R_2$ naturally faces the problem of choosing which relation to sample. We argue that we can choose either R_1 or R_2, as long as we also choose the appropriate sample size as described in the Similarity Sampling section. For a specific instance of the problem, we can break this asymmetry by executing the approximate join twice. Thus, we first sample from vectors of R_2 and use R_1 to weight the samples. Then, we sample from vectors of R_1 and use R_2 to weight the samples. Then, we take

the union of these as our final result. We refer to this as a symmetric text-join. We will evaluate this technique experimentally in the Experimental Evaluation. In this section we have showed how to approximate the text-join $R_1 \bowtie_\phi R_2$ by using weighted sampling .In the next section, we describe how this approximate join can be completely implemented using a standard, unmodified RDBMS.

Sampling and Joining Tuple Vectors in SQL

We now describe an SQL implementation of the sampling-based join algorithm of the previous section. There is first described the Sampling step, and then focuses on the Weight and Thresholding steps for the asymmetric versions of the join. Finally, the implementation of a symmetric version of the approximate join is described.

Implementing the Sampling Step in SQL

Given the R_i*Weights* relations, we now show how to implement the Sampling step of our text-join approx-imation strategy in SQL. For a desired sample size S and similarity threshold ϕ, we create the auxiliary relation shown in Figure 3. As the SQL statement in the figure shows, we join the relations *RiWeights* and *RiSum* on the token attribute. The P attribute for a tuple in the result is the probability;

$$\frac{RiWeights.weight}{RiSum.total}$$

with which we should pick this tuple. Conceptually, for each tuple in the output of the query of Figure 3 we need to perform S trials, picking each time the tuple with probability P.For each successful trial, we insert the corresponding tuple (*tid, token*) in a relation *RiSample* (*tid, token*),preserving duplicates. The SQL statement utilizes a relation R1V to implement the

Weight step, storing the $T_v(t_q)$ values for each tuple $t_q \in R_1$. As described later, the R1V relation can be eliminated from the query and is just shown here for clarity. The *S* trials can be implemented in various ways. One (expensive) way to do this is as follows: We add "AND P $\geq$ RAND()" in the WHERE clause of the Figure 3 query, so that the execution of this query corresponds to one "trial." Then, executing this query *S* times and taking the union of the all results provides the desired answer. A more efficient alternative, which is what we implemented, is to open a cursor on the result of the query in Figure 3, read one tuple at a time, perform *S* trials on each tuple, and then write back the result. Finally, a pure-SQL "simulation" of the Sampling step deterministically defines that each tuple will result in;

$$Round(S \cdot \frac{R1Weights.weight}{R1Sum.total})$$

"successes" after *S* trials, on average. This deterministic version of the query is shown in Figure 4. We have implemented and run experiments using the deterministic version, and obtained virtually the same performance as with the Cursor-based implementation of sampling over the Figure 3 query. In the remainder of this description, in order to keep the discussion close to a probabilistic framework a cursor-based approach for the Sampling step is used.

Implementing the Weight and Thresholding Steps in SQL

The Weight and Thresholding steps are previously described as two separate steps. In practice, we can combine them into one SQL statement, shown in Figure 5. The Weight step is implemented by the SUM aggregate in the "HAVING" clause". We weight each tuple from the sample according to;

$$\frac{R1Weights.weight \cdot R2Sum.total}{R1V.T_V},$$

Then, we can count the number of times that each which corresponds to;

$$\frac{v_{t_q}(j) \cdot Sum(j)}{Tv(t_q)}$$

The we can count the number of times that each particular tuple pair appears in the results (see GROUP BY clause). For each group, the result of the SUM is the number of times *C*; that a specific tuple pair appears in the candidate set. To implement the Thresholding step, we apply the count filter as a simple comparison in the HAVING clause: we check whether the frequency of a tuple pair exceeds the count threshold (i.e.;

$$(\text{i.e., } C_i > \frac{s}{Tv(t_q)}\phi')$$

The final output of this SQL operation is a set of tuple id pairs with expected similarity exceeding threshold ϕ. The SQL statement in Figure 5 can be further simplified by completely eliminating the join with the R1V relation. The *RIV.Tv* values are used only in the HAVING clause, to divide both parts of the inequality. The result of the inequality is not affected by this division, hence the R1V relation can be eliminated when combining the *Weight* and the *Thresholding* step into one SQL statement.

Implementing a Symmetric Text-Join Approximation in SQL

Up to now we have described only an asymmetric text-join approximation approach, in which we sample relation R_2 and weight the samples according to the tuples in R_1 (or vice versa). However, as we described previously, the text-join $R_1 \tilde{\bowtie}_\phi R_2$ treats R_1 and R_2 symmetrically. To break the asymmetry of our sampling-based strategy, we execute the two different asymmetric approximations and report the union of their results, as shown in Figure 6. Note that a tuple pair *(tid1, tid2)* that appears in the result of the two intervening asymmetric approximations needs

high combined "support" to qualify in the final answer (see HAVING clause in Figure 6).An additional strategy naturally suggests itself: Instead of executing the symmetric join algorithm by joining the samples with the original relations, we can just join the samples, ignoring the original relations. This version of the sampling-based text-join makes an independence assumption between the two relations. We sample each relation independently, join the samples, and then weight and threshold the output. We implement the Weight step by weighting each tuple with

$$\frac{R1.Sum.total}{R1V.T_V} \cdot \frac{R2.Sum.total}{R2V.T_V}.$$

The count threshold in this case becomes;

$$C_i > \frac{S \cdot S}{T_V(t_q) \cdot T_V(t_i)} \phi'$$

(again the T_v values can be eliminated from the SQL if we combine the *Weight* and the *Thresholding* steps). Figure 7 shows the SQL implementation of this version of the sampling-based text-join.

Experimental Evaluation

We implemented the proposed techniques and performed a thorough experimental evaluation in terms of both accuracy and performance. We first describe the techniques that we compare and the data sets and metrics that we use for our experiments. Then, we report the experimental results.

Experimental Settings

The schema and the relations described in Creating Weight Vectors for Tuples, were implemented on a commercial RDMBS, MicrosoftSQL Server 2000, running on a 550 MHz Pentium III-based PC with 768Mb of RAM. SQL Server was configured to potentially utilize the entire RAM as a buffer pool.

Data Sets: For our experiments, we used real data from an AT&T customer relationship database. We extracted from this database a random sample of 40,000 distinct attribute values of type string. We then split this sample into two data sets, R_1 and R_2. Data set R_1 contains about 14,000 strings, while data set R_2 contains about 26,000 strings. The average string length for R_1 is 19 characters and, on average, each string consists of 2.5 words. The average string length for R_2 is 21 characters and, on average, each string consists of 2.5 words. The length of the strings follows a close-to-Gaussian distribution for both data sets and is reported in Figure 8(a), while the size of;

$R_1 \boxtimes_\phi R_2$ for different similarity thresholds ϕ and token choices is reported in Figure 8(b).

Metrics: To evaluate the accuracy and completeness of our techniques we use the standard precision and recall metrics:

Definition 3 *Consider two relations R_1 and R_2 and a user-specified similarity threshold ϕ. Let $Answer_\phi$ be an approximate answer for text-join $R_1 \boxtimes_\phi R_2$. Then, the* precision *and* recall *of $Answer_\phi$ with respect to $R_1 \boxtimes_\phi R_2$ are defined as:*

$$precision = \frac{|Answer_\phi \cap (R_1 \boxtimes_\phi R_2)|}{|Answer_\phi|} \quad and \quad recall = \frac{|Answer_\phi \cap (R_1 \boxtimes_\phi R_2)|}{|R_1 \boxtimes_\phi R_2|}$$

Precision and recall can take values in the 0-to-1 range. Precision measures the accuracy of the answer and indicates the fraction of tuples in the approximation of;

$R_1 \boxtimes_\phi R_2$

that are correct. In contrast, recall measures the completeness of the answer and indicates the fraction of the;

$R_1 \boxtimes_\phi R_2$

tuples that are captured in the approximation. For data cleaning applications, we believe that recall is more important than precision. The returned answer can always be checked for false

positives in a post-join step, while we cannot locate false negatives without re-running the text-join algorithm. Finally, to measure the efficiency of the algorithms, we measure the actual execution time of the similarity join for different techniques.

Techniques Compared:

We compare the following algorithms for computing (an approximation of);

$R_1 \bowtie_\phi R_2$

All of these algorithms can be deployed completely within an RDBMS:

- *Baseline*: This expensive algorithm (Figure 2) computes the exact answer for $R_1 \bowtie_\phi R_2$ by considering all pairs of tuples from both relations.
- *R1sR2*: This asymmetric approximation of $R_1 \bowtie_\phi R_2$ samples relation R_2 and weights the sample using R_1 (Figure 5).
- *sR1R2*: This asymmetric approximation of $R_1 \bowtie_\phi R_2$ samples relation R_1 and weights the sample using R_2.
- *R1R2*: This symmetric approximation of $R_1 \bowtie_\phi R_2$ is shown in Figure 6.
- *sR1sR2*: This symmetric approximation of $R_1 \bowtie_\phi R_2$ joins the two samples *R1Sample* and *R2Sample* (Figure 7).

In addition, we also compare the SQL-based techniques against the stand-alone WHIRL system. Given a similarity threshold ϕ and two relations R_1 and R_2, WHIRL computes the text-join

$R_1 \bowtie_\phi R_2$

The fundamental difference with our techniques is that WHIRL is a separate application, not connected to any RDBMS. Initially, we attempted WHIRL over our data sets using its default settings. Unfortunately, during the computation of the

$R_1 \bowtie_\phi R_2$

join WHIRL ran out of memory. We then limited the maximum heap size 6 to produce an approximate answer for

$R_1 \bowtie_\phi R_2$

We measure the precision and recall of the WHIRL answers, in addition to the running time to produce them. Choice of Tokens: We present experiments for different choices of tokens for the similarity computation. The token types that we consider in our experiments are:

• *Words*: All space-delimited words in a tuple are used as tokens (e.g., "AT&T" and "Labs" for string "AT&T Labs").

• *Q-grams*: All substrings of q consecutive characters in a tuple are used as tokens (e.g., "$A," "AT&T&," "&T," "T ," " L," "La," "ab," "bs," "s#," for string "AT&T Labs" and $q = 2$, after we append dummy characters "$" and "#" at the beginning and end of the tuple). We consider $q = 2$ and $q = 3$.

The *RiWeights* table has 30,933 rows for Words, 268_458 rows for *Q-grams* with $q = 3$, and 245,739 rows for *Q-grams* with $q = 2$. For the *R2Weights* table, the corresponding numbers of rows are 61,715, 536,982,and 491_515. In Figure 8(b) we show the number of tuple pairs in the exact result of the text-join;

$R_1 \bowtie_\phi R_2$,

for the different token choices and for different similarity thresholds;

ϕ.

Unfortunately, WHIRL natively supports only word tokenization but not *q-grams*. To test WHIRL with *q-grams*, we adopted the following strategy: We generated all the *q-grams* of the strings in R_1 and R_2, and stored them as separate "words." For example, the string "ABC" was transformed into "$A ABBC C#" for $q = 2$. Then WHIRL used the transformed data set as if each q-gram were a separate word. Besides the specific choice of tokens, three other main parameters affect the performance and accuracy of our techniques: the sample size S, the choice of the user-defined similarity threshold

ϕ, and the choice of the error margin ϵ. We now experimentally study how these parameters affect the accuracy and efficiency of sampling-based text-joins.

Experimental Results

Comparing Different Techniques: Our first experiment evaluates the precision and recall achieved by the different versions of the sampling-based text-joins and for WHIRL (Figure 9). For sampling-based joins, a sample size of $S = 128$ is used (we present experiments for varying sample size S below).Figure 9(a) presents the results for Words and Figures 9(b)(c) present the results for *Q-grams*, for $q = 2$ and $q = 3$. WHIRL has perfect precision (WHIRL computes the actual similarity of the tuple pairs),but it demonstrates very low recall for *Q-grams*. The low recall is, to some extent, a result of the small heap size that we had to use to allow WHIRL to handle our data sets. The sampling-based joins, on the other hand, perform better. For Words, they achieve recall higher than 0.8 for thresholds $\phi > 0.1$, with precision above 0.7 for most cases when $\phi > 0.2$ (with the exception of the *sRlsR2* technique). WHIRL has comparable performance for $\phi > 0.5$. For *Q-grams* with $q = 3$, *sR1R2* has recall around 0.4 across different similarity metrics, with precision consistently above 0.7, outperforming WHIRL in terms of recall across all similarity thresholds. When q = 2, none of the algorithms performs well. For the sampling-based text-joins this is due to the small number of different tokens for q = 2. By comparing the different versions of the sampling-based joins we can see that *sRlsR2* Performs worse than the other techniques in terms of precision and recall. Also, *RlsR2* is always worse than *sRIR2*: Since R_2 is larger than R_1 and the sample size is constant, the sample of R_1 represents the R_1 contents better than the corresponding sample of R_2 does for R_2.

Effect of Sample Size S:

The second set of experiments evaluates the effect of the sample size

As we increase the number of samples S for each distinct token of the relation, more tuples are sampled and included in the final sample. This results in more matches in the final join, and, hence in higher recall. It is also interesting to observe the effect of the sample size for different token choices. The recall for *Q-grams* with $q = 2$ is smaller than that for *Q-grams* with $q = 3$ for a given sample size, which in turn is smaller than the recall for Words. Since we independently obtain a constant number of samples per distinct token, the higher the number of distinct tokens the more accurate the sampling is expected to be. This effect is visible in the recall plots of Figure 10. The sample size also affects precision. When we increase the sample size, precision generally increases. However, in specific cases we can observe that smaller sizes can in fact achieve higher precision. This happens because for a smaller sample size we may get an underestimate of the similarity value (e.g., estimated similarity 0.5 for real similarity 0.7).Underestimates do not have a negative effect on precision. However, an increase in the sample size might result in an overestimate of the similarity, even if the absolute estimation error is smaller (e.g., estimated similarity 0.8 for real similarity 0.7). Overestimates, though, affect precision negatively when the similarity threshold ϕ happens to be between the real and the (over)estimated similarity.

Effect of Error Margin ϵ:

As mentioned in previously, the threshold for count filter is;

$$\frac{S}{Tv(t_q)}(1-\epsilon)\phi.$$

Different values of ϵ affect the precision and recall of the answer. Figure 11 shows how different choices of ϵ affect precision and recall. When we increase ϵ, we lower the threshold for count filter and more tuple pairs are included in the answer. This, of course, increases recall, at the expense of precision: the tuple pairs included in the result have estimated similarity lower than the desired threshold ϕ. The choice of ϵ is an "editorial" decision, and should be set to either favor recall or precision. As discussed above, we believe that higher recall is more important for data cleaning applications. The returned answer can always be checked for false positives in a post-join step, while we cannot locate false negatives without re-running the text-join algorithm.

Execution Time:

To analyze efficiency, we measure the execution time of the different techniques. Our measurements do not include the preprocessing step to build the auxiliary tables in Figure 1: This preprocessing step is common to the baseline and all sampling-based text-join approaches. This preprocessing step took less than two minutes to process both relations R_1 and R_2 for Words, and about five minutes for *Q-grams*. Also, the time needed to create the *RiSample* relations is less than five seconds. For WHIRL we similarly do not include the time needed to export the relations from the RDBMS to a text file formatted as expected by WHIRL, the time needed to load the text files from disk, or the time needed to construct the inverted indexes 7. The preprocessing time for WHIRL is about 15 seconds for Words and one minute for *Q-grams*, which is smaller than for the sampling-based techniques: WHIRL keeps the data in main memory, while we keep the weights in materialized relations inside the RDBMS. The Baseline technique (Figure 2) could only-be run for Words. For *Q-grams*, SQL Server executed the Baseline query for approximately 7 hours before finishing abnormally. Hence, we only report

results for Words for the Baseline technique. Figure 12(a) reports the execution time of sampling-based text-join variations for Words, for different sample sizes. The execution time of the join did not change considerably for different similarity thresholds, and is consistently lower than that for Baseline. The results for Figure 12 were computed for similarity threshold, ϕ=0.5; the execution times for other values of ϕ are not significantly different. For example, for $S = 64$, a sample size that results in high precision and recall (Figure 10(a)), *R1R2* is more than 10 times faster than Baseline. The speedup is even higher for *sR1R2* and *RlsR2*. Figures 12(b) and 12(c) report the execution time for *Q-grams* with $q = 2$ and $q = 3$. Not surprisingly, *sRlsR2*, which joins only the two samples, is considerably faster than the other variations.

Similarity Function	*Mismatches Captured*	*Mismatches not Captured*
Edit distance	Spelling errors, insertions and deletions of short words	Variations of word order, insertions and deletions of long words
Block edit distance	Spelling errors, insertions and deletions of short words, variations of word order	Insertions and deletions of long words
Cosine similarity with words as tokens	Insertions and deletions of common words, variations of word order	Spelling errors
Cosine similarity with q-grams as tokens	Spelling errors, insertions and deletions of short or common words, variations of word order	-

Table 1: Different similarity functions for data cleansing, and the types of string mismatches that they can capture.

This faster execution, however, is at the expense of accuracy (Figure 9). For all choices of tokens, the symmetric version *RIR2* has an associated execution time that is longer than the sum of the execution times of *sR1R2* and *RlsR2*. This is expected, since *RIR2* requires executing, *sRIR2* and *RlsR2* to compute its answer. Finally, Figure 12(d) lists the execution time for WHIRL, for different similarity thresholds. For *Q-grams* with $q = 3$, the execution time for WHIRL is roughly comparable to that of *RlsR2* when $S = 128$. For this setting *RlsR2* has recall generally at or above 0.2, while WHIRL has recall usually lower than 0.1. For Words, WHIRL is more efficient than the sampling-based techniques for high values of S, while WHIRL has

significantly lower recall for low to moderate similarity thresholds (Figure 9(a)). For example, for $S = 128$ sampling-based text-joins have recall above 0.8 when;

$$\phi > 0.1$$

and WHIRL has recall above 0.8 only when;

$$\phi > 0.5.$$

In general, the sampling-based text-joins, which are executed in an unmodified RDBMS, have efficiency comparable to WHIRL, provided that WHIRL has sufficient main memory available: WHIRL is a stand-alone application that implements a main-memory version of the A* algorithm. This algorithm requires keeping large search structures during processing; when main memory is not sufficiently large for a dataset, WHIRL's recall suffers considerably. In contrast, our techniques are fully executed within RDBMSs, which are specifically designed to handle large data volumes in an efficient and scalable way.

Using Different Similarity Functions for Data Cleansing

The Experimental Evaluation studied the accuracy and efficiency of the proposed sampling-based text-join executions according to the present invention, for different token choices and for a distance metric based on *tf.idf* token weights. We now compare this distance metric against string edit distance, especially in terms of the effectiveness of the distance metrics in helping data cleansing applications. The edit distance between two strings is the minimum number of edit operations (i.e., insertions, deletions, and substitutions) of single characters needed to transform the first string into the second. The edit distance metric works very well for capturing typographical errors. For example, the strings "ComputerScience" and "Computer Science" have edit distance one. Also edit distance can capture insertions of short words (e.g., "Microsoft" and

"Microsoft Co" have edit distance three). Unfortunately, a small increase of the distance threshold can result in many false positives, especially for short strings. For example, the string "IBM" is within edit distance three of both "ACM" and "IBM Co. "The simple edit distance metric does not work well when the compared strings involve block moves (e.g., "Computer Science Department" and "Department of Computer Science"). In this case, we can use block edit distance, a more general edit distance metric that allows for block moves as a basic edit operation. By allowing for block moves, the block edit distance can also capture word rearrangements. Finding the exact block edit distance of two strings is an NP-hard problem. Block edit distance cannot capture all mismatches. Differences between records also occur due to insertions and deletions of common words. For example, "KAR Corporation International" and "KAR Corporation" have block edit distance 14. If we allow large edit distance threshold capture such mismatches, the answer will contain a large number of false positive matches. The insertion and deletion of common words can be handled effectively with the cosine similarity metric that we have described in this paper if we use words as tokens. Common words, like "International," have low *idf* weight. Hence, two strings are deemed similar when they share many identical words (i.e., with no spelling mistakes) that do not appear frequently in the relation. This metric also handles block moves naturally. The use of words as tokens in conjunction with the cosine similarity as distance metric was proposed by WHIRL. Unfortunately, this similarity metric does not capture word spelling errors, especially if they are pervasive and affect many of the words in the strings. For example, the strings "Computer Science Department" and "Department of Computer Science" will have zero similarity under this metric. Hence, we can see that (block) edit distance and cosine similarity with words serve complementary purposes for data cleansing applications. Edit distance handles spelling errors

well (and possibly blockmoves as well), while the cosine similarity with words nicely handles block moves and insertions of words. A similarity function that naturally combines the good properties of the two distance metrics is the cosine similarity with *q-grams* as tokens. A block move minimally affects the set of common *q-grams* of two strings, so the two strings "Gateway Communications" and "Communications Gateway" have high similarity under this metric. A related argument holds when there are spelling mistakes in these words. Hence, "Gateway Communications" and "Communications Gateway" will also have high similarity under this metric despite the block move and the spelling errors in both words. Finally this metric handles the insertion and deletion of words nicely. The string "Gateway Communications" matches with high similarity the string "Communications Gateway International" since the *q-grams* of the word "International" appear often in the relation and have low weight. Table 1 summarizes the qualitative properties of the distance functions that we have described in this section. The choice of similarity function impacts the execution time of the associated text-joins. The use of the cosine similarity with words leads to fast query executions as we have seen in the Experimental Evaluation. When we use *q-grams*, the execution time of the join increases considerably, resulting nevertheless in higher quality of results with matches that neither edit distance nor cosine similarity with words could have captured. Given the improved recall and precision of the sampling-based text join when $q = 3$ (compared to the case where $q = 2$), we believe that the cosine similarity metric with 3-grams can serve well for data cleansing applications.

It will be appreciated that the present invention has been described herein with reference to certain preferred or exemplary embodiments. The preferred or exemplary embodiments described herein may be modified, changed, added to or deviated from without departing from the intent, spirit and scope of the present invention. It is intended that all such additions,

modifications, amendments, and/or deviations be included within the scope of the claims appended hereto.